A "Framework" for Object Oriented Frameworks Design

Parsons, David; Rashid, Awais; Speck, Andreas; Telea, Alexandru

Published in:
EPRINTS-BOOK-TITLE

A "Framework" for Object Oriented Frameworks Design

David Parsons[1], Awais Rashid[2], Andreas Speck[3], Alexandru Telea[4]
[1] Systems Engineering Faculty, Southampton Institute, UK
[2] Department of Computer Science, Lancaster University, UK
[3] Faculty of Computing Science, University of Tuebingen, Germany
[4] Department of Mathematics and Computing Science,
Eindhoven University of Technology, The Netherlands

Abstract

Object-oriented frameworks are established tools for domain-specific reuse. Many framework design patterns have been documented, e.g. reverse engineering framework architectures from conventionally built applications for a given domain. The framework development cycle generally evolves from open framework to closed application. We describe a more flexible component-based approach to framework design that stresses a common interface for 'plugging-in' new components at different lifecycle stages. An analysis of framework-related user roles shows that the classical developer/end-user boundary is too rigid. We see the framework's development as a continuum within which its 'actors' can customise its behaviour. This both increases the system's flexibility and reduces its maintenance requirement. A case study of three frameworks for different application domains illustrates the presented principles.

1: Introduction

A central tenet of object technology adoption has been the reuse promise, but this has proved difficult to deliver in practice. Inheritance-based reuse has been effective only in limited cases, such as general purpose components as container There is a trade-off between reusability and tailorability [3], as the user's requirements cannot be effectively anticipated. In practice, reuse can only be achieved within domain-specific constraints. The most effective route to reuse has been that of the (object-oriented) *framework*, where the reuse context is constrained to a given domain. A framework can be defined as "a system that can be customised, specialised, or extended to provide more specific, more appropriate, or slightly different capabilities" (see Gabriel [4]). Application-specific frameworks cover precise, focussed domains (e.g. hardware control systems [10], scientific visualisation and simulation [14, 15, 19], and thus are highly reusable. Frameworks consist of *frozen spots* (already coded software pieces to be reused) and *hot spots* (flexible elements, allowing users to adjust the framework to concrete application needs). Unlike most class libraries, frameworks encapsulate control flows as well as object interfaces, thus modelling a system's dynamic behaviour as well as its structure.

If we accept frameworks as effective for domain-specific reuse, then we might ask ourselves what is a framework architecture and how can we develop an effective framework that meets its users' needs. In this paper we discuss the frameworks' characteristics, the roles and interactions of their developers and users, and catalogue the framework development issues and trade-offs. We propose a component-driven approach to framework design that stresses system flexibility at all development, use, and maintenance phases. User requirements analysis is seen as as an essential aspect of the framework designer's task, since meeting different requirements in different domains can often lead to radically different framework designs. We group

framework users into a number of roles, each requiring a suitable component level interface. The requirements' union is seen as the interface specification that the framework designer must provide. We illustrate our "framework" for frameworks design by three frameworks for different application domains, built on its principles.

2: Flexible Elements of Object Oriented Frameworks

Based on customisation characteristics, OO frameworks fall into two main categories:

- **White Box Frameworks** Component and application developers need to know the white box frameworks' architecture to adapt the framework to a concrete application. The hot spots are usually limited to *inheritance* (Fig. 1 left). When a system's hot spots are clear, building a white box framework (e.g. by generalising from a few complete applications) is relatively easy. The disadvantage of such frameworks is that the end user needs to know about the complete architecture in order to use it, implying a long learning curve and high error risks. Many OO application libraries are build as white box frameworks [15, 13].
- **Black Box Frameworks** Black box frameworks, in contrast, hide their internal structure. Users just know a general framework description and its hot spots rather than detail architecture knowledge. Using black box frameworks is thus easier than using white box frameworks. Black box frameworks, however, are harder to build than white box frameworks. The hot spot mechanism is usually *composition* (Fig. 1 left). Black box frameworks implement the information hiding principle of Parnas [7] best. They are often found as application specific frameworks (e.g. for machinery controlling [10], scientific visualisation [14] or data processing [5]) where software components mirror application domain concepts. Black box frameworks are harder to build for less domain-specific cases as it is more difficult to anticipate the required component types.

Few 'pure' white or black box frameworks exist in practice. Often some hot spots are developed using a white box approach while others use the black box approach. During framework implementation white box elements can be refined to black box ones. Frameworks tend to mature in this process, beginning as white box and evolving to black box frameworks. Our component driven approach to framework development supports both white and black box frameworks. The framework components may contain both elements in arbitrary combination.

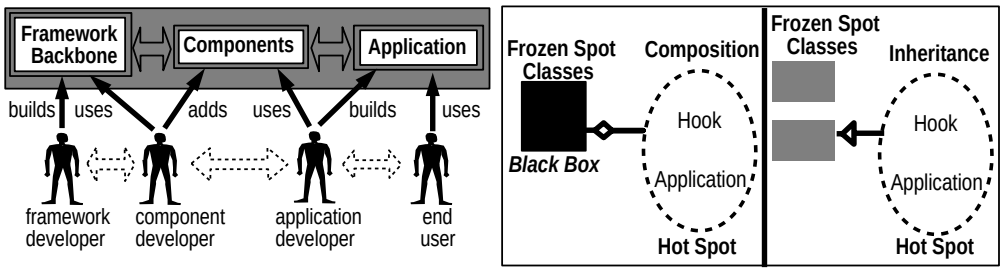


Figure 1. Hot spots in black box and white box frameworks (left) Framework user roles (right)

3: Actors and Their Roles

During framework-based application development, users fall into four categories or*roles*: framework developer (FD), component developer (CD), application developer (AD), and end user (EU) (Fig. 1 right, the solid arrows show the actions involving these roles). The FD builds the framework 'backbone'. The CD builds new components and integrates them in the framework. The AD assembles these components into a custom application expected by the EU. Consider e.g. a graphical user interface (GUI) framework built by the FD, extended by CDs with add-on GUI components, and used by ADs to build applications for EUs. Software reuse appears in two places. First, CDs can reuse code via constructs of the component development language (inheritance, templates, etc) to build new components. Second, the AD employs the framework's specific 'super language' reuse mechanisms to reuse components when building applications or even to build components out of existing ones [6]. The latter mechanisms are more effective/flexible than the former, since designed for a specific application domain. The dashed block arrows in Fig. 1 right show that the presented roles may be assumed by the same people. For example, although the FD is mainly responsible for building the framework, he may also be the first CD, e.g. in frameworks as OLE, CORBA implementations and Java Beans. Similarly the CDs, ADs and the EUs could be the same people. The FD's core task is to provide tools for all actors (e.g. component building tools for the CDs, application assembly tools for ADs, and possibly command and control interfaces for EUs). Even though developers and end users may (and should be able to) perform their specific tasks independently the FD must implement *all* the mechanisms for *all* these roles, such that each role gets the best tools it requires. In addition to providing a framework satisfying all roles separately, FDs must provide for an easy role transition, since the same person may frequently switch roles. For example a scientist writes components as a CD, then assembles them into a test application as an AD, and finally experiments with the application as an EU.

4: A Component-Driven Approach To Framework Design

To satisfy the previously presented requirements, we advocate a framework architecture based on highly cohesive but loosely coupled *components*, managed by a central framework *backbone* (Fig 2 left). Components can then be developed independently of each other and are exchangeable hence providing an overall high flexibility degree. The backbone is the core element; its design depends strongly on the target application domain. It mainly provides communication, data exchange, and synchronisation mechanisms, and hot spots for plugging-in components. Basic components contain the basic framework functionality and are built by the FD or the CD. Through the *component-only-customisable hot spots* basic components use the backbone for data exchange and communication. The framework backbone and the basic components are mandatory for a complete framework. Component or application developers can customise the components but not the backbone, as the latter should maintain the modelled domain's invariants. Additional components may be further added by the CD or AD to adapt the framework to additional requirements. They may interact with the basic components and with the backbone, that calls back on their services. This modular framework architecture allows the backbone and also the components' reuse in other contexts. To build such a component-based framework which satisfies the user requirements listed in Section 3, we propose a methodology having the following four steps.

- **1. Application Domain and Role Definition** First we identify the application domain to be covered by the framework. The different user requests are classified into user roles, whose requirements and mutual interactions are clearly defined. These

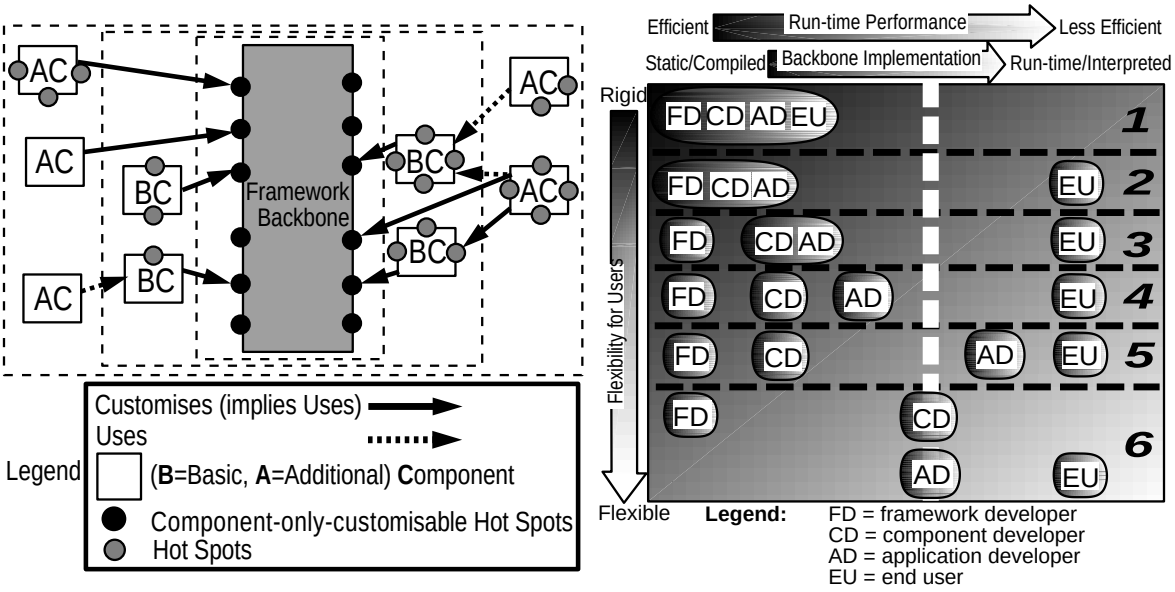


Figure 2. Framework backbone with plugged-in components (left) Framework implementation trade-offs (right)

determine the framework design requirements, which outline a base for the development [11]. How user requirements analysis influences the framework implementation (e.g. choosing an OO design or not, how we define the hot spots, etc) is detailed in Section 5. This is the crucial phase of the whole process, as a correct definition of roles and requirements strongly influences all the following steps.

- **2. Backbone Development** The framework backbone is designed and implemented here by the FD as a logical conclusion of the already found requirements. This phase is also detailed in Section 5.
- **3. Basic Components Development** The basic components and their hot spots are now developed, in conformance with the standard component interface requirement implied by the existing backbone. This conformance should not constrain the CD, as these interfaces were inferred directly from the users' requirements in the first phase.
- **4. Additional Components Development** Finally, CDs or ADs may create an open set of custom components which directly plug in the existing framework and may or may not use the basic components. The only constraint is that additional components respect the backbone's interfaces, which is ensured by the requirement analysis in the first phase.

The above phases have no strict boundary like in the waterfall model [2]. As in other OO software development processes (e.g. [1]) the borders between phases are weak and designers may work in two phases on different components at the same time. However a careful requirement analysis in phase 1 will yield a stable framework, such that redesign gets localised to phases 3 and 4.

5: Backbone Architecture and Flexibility Requirements

As outlined in the previous section, providing the right flexibility and user requirements' fulfilment inferred from the roles' analysis in the backbone differentiates a successful and a rigid, inadequate framework. Possible implementations of the presented component-based

framework model can be seen as a continuum between fully *compiled* and fully *interpreted* systems. This viewpoint is very important as the freedom offered to the user roles, the backbone design complexity and the system's overall performance strongly depend on the compiled/interpreted ratio used in the framework's design. Figure 2 right depicts this continuum: every horizontal band corresponds to a framework design type; for a given design, the users' roles (FD,CD,AD,EU) are placed with respect to the compiled or interpreted techniques used for the implementation of their requirements. In practice, we distinguish six main framework implementation classes, as follows.

1. Monolithic systems

This is the traditional style of programming an application as a monolith in which all requirements are addressed and resolved on the same implementation level. There is no distinction between backbone and components, or between the users' roles. The resulting systems (hardly deserving the framework name) are very execution-efficient, straightforward to program (usually a single executable written in a compiled language) but must be constantly redesigned/recoded to account for new requests. They are often designed, maintained and used by the same person.

2. Modular Systems

These systems separate the EU from the FD. The former can directly steer the system via some interface or command language without knowing its implementation details, but he still relies on the latter to redesign, recode (and recompile) the system to account for new requirements. Most GUI-based / configuration file driven applications fall in this case.

3. Application Libraries

The first step towards reuse takes the form of application libraries, which are designed once by the FD (mostly in compiled form) and used by ADs to produce different applications for the EUs. Object-oriented techniques increase the reusability of such libraries. There is usually no distinction between the CD and AD, as they represent the same person using the library.

4. Compiled Component-Based Frameworks

As application libraries keep growing e.g. by adding new classes, designers notice that these can be separated into a reusable *fixed part*, coding the domain-dependent infrastructure and possibly some control logic, and a *variable part*, i.e. the new classes added to satisfy the users' increasing demands. The fixed part becomes the framework backbone, designed once, by the FD. The variable part contains the open set of classes raised to the rank of *components*, as they get a more formal (but also more flexible) interface for interaction with the backbone. The CD role emerges as a mediator between the fixed backbone and the AD's variable demands. In the simplest version, such frameworks come as compiled applications: components can be designed relatively easily, but a running application needs recompilation to change existing components or add new ones.

5. Dynamic Component-Based Frameworks

Dynamic component frameworks extend compiled frameworks by allowing dynamic loading of component types. Components are still not modifiable in the running application,

but new ones can be designed off-line and loaded in the (already running) framework. The AD role is emphasized by the appearance of *application development environments*, in which applications can be built interactively by visually assembling precompiled components. Examples range from simulation and visualisation [14, 15, 19] to data processing systems [5]. Applications are often no longer compiled, but interpreted in a run-time language which calls back on the compiled components. The main problem of such dual-language frameworks is that the component development (compiled) language often differs from the application (interpreted) language. This makes difficult to map constructs in one language to the other, so CDs may have difficulties passing their work to the ADs. Several ad-hoc techniques are used for interfacing the two languages (practically designing the system's hot spots), mainly trying to simulate run-time typing and reflection using components created in a compile-time typed language. Among these we note Java, tcl [15] or Objective C class wrappers for C++ [19], exemplar implementations, adapter classes generating parallel hierarchies [18], run-time type information, etc.

6. Single Language Frameworks

Single language frameworks remove most of the problems of dual-language ones. The component development language coincides with the application specification one, so the framework directly accepts the supplied components (the hot spots do not have to do a language mapping). CDs and ADs can often use the same language in interpreted mode to dynamically define types and build applications at run-time and features like dynamic component loading or just in time compilation, so the role transition becomes easier. Examples of single language frameworks are Java Beans based systems, the ROOT data processing system [5], or the development environment described by Meyer in [17]. Extreme examples are frameworks based on fully interpreted or typeless languages, like Smalltalk or Lisp. These incur however performance problems and are not attractive for CDs having to reuse compiled (e.g. C/C++/Fortran) legacy code.

6: A Case Study on Component Based Frameworks

Choosing the right backbone implementation out of the ones presented in the previous section crucially determines how the framework will meet the users' expectations. We conclude our "framework" for framework design by presenting three frameworks with different architectures, emerged from the analysis of different application domains and user requirements on the model presented so far.

6.1 A Framework For Schematic Capture

This framework for electronic design automation (EDA) converts graphical representations of electronic circuit designs into a hardware description language for analogue and mixed signal (VHDL-AMS). The schematic capture system allows the CD to model new component types. This is usually done by by aggregation (building larger components from smaller ones) but languages such as VHDL-AMS also allow behavioural modelling, where components can be described in terms of code-specified behaviour. Circuit designers using a schematic capture system thus need a facility for describing new components both behaviourally (in VHDL code) and visually (using GUI tools, see Fig. 4). In our framework, new components are described dynamically at run time. The three key elements of the system are:

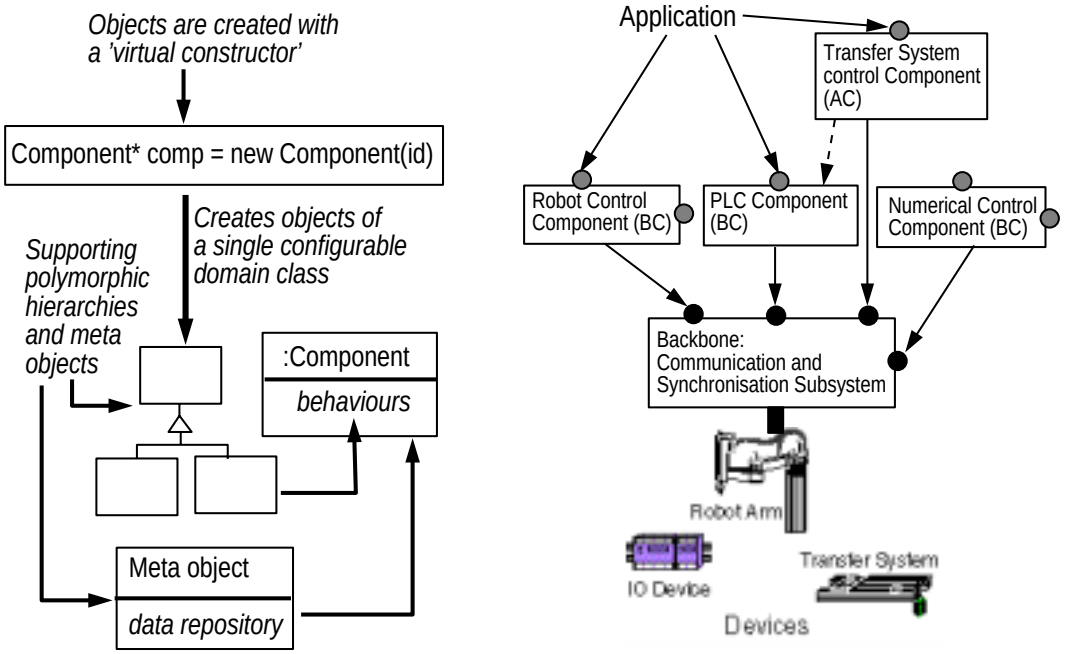
- **Framework Backbone:** this provides the means for instantiating and connecting objects representing electronic components, and the core algorithms for circuit analysis

and VHDL code generation.

- **Basic Components:** these are the basic set of electronic components used by most designers (e.g. various digital gates and standard analogue components such as resistors and capacitors).
- **Additional Components:** these are provided at run-time by the user who can describe them via a visual object building GUI providing the context for behavioural modelling.

Referring to the development phases in Section 4, we identify the following steps:
1. Identification of the requirements a schematic capture system within domain of EDA.
2. Framework backbone development, including the hot spots where component objects plug in. This includes adding visual images to the library and appropriate syntax for code generation.
3. Development of the basic library of electronic components
4. Development of additional electronic component models using a run time extensibility mechanism. Since this mechanism is also available at compile time, these components can be developed by the CD, the AD or the EU (these roles overlap).

The system supports dynamic extensibility via a standard 'Component' interface supported by meta data that allows flexible component configuration. There is no need to compile the new classes, as the run time system interprets configuration data dynamically to provide different behaviours for different component types. The Component interface (Fig. 3 a) is provided via a single class rather than an inheritance hierarchy. This is though supported



(a) Schematic capture framework **(b) Industrial control framework**

Figure 3.

by other polymorphic classification hierarchies at a lower granularity level, configured by data provided by another object. Since this data can be changed and extended dynamically, component instances can be provided with new behaviours at run time. Summarising, the need of run-time, flexible configurability determines a single-language, dynamic architecture (type 6 in Section 4).

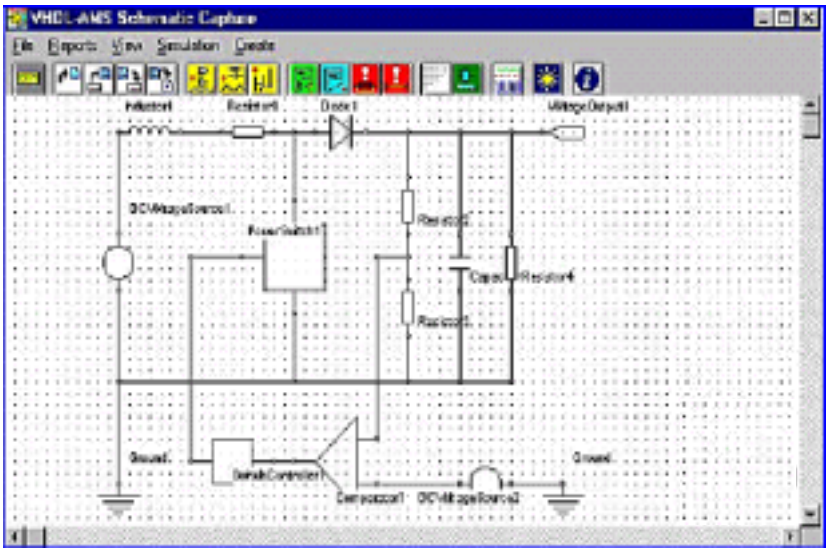


Figure 4. Electronic design automation framework GUI

6.2 An Interactive Scientific Simulation and Visualisation Framework

Insight in complex physical processes requires integration of scientific visualisation and numerical simulation in a single interactive framework. The presented framework [16] provides simulation description, parameter input, computation, and visualisation phases. ADs can build simulations visually by assembling a set of icons (Fig. 5 a) representing user-programmable OO components in a data-flow network (Fig. 5 b) EUs steer these applications by several GUIs (Fig. 5 f). and visualise the results interactively (Figs. 5 c,d,e) show several medical imaging visualisations).

- **Framework Components:** the component notion extends the C++ class concept with dataflow semantics, adding notions as inputs, outputs (through which data is exchanged with other connected components), and an update method (called by the backbone when the component inputs' change). CDs write C++ classes and then interface them with the framework by writing a set of *metaclasses* (OO entities in a simple declarative meta-language actually being the components). This makes any C++ class hierarchy integrable in the framework without adapting its code at all. There are no basic and additional components, but several component libraries for several application areas (scientific visualisation, finite element analysis, image processing, 3D graphics, etc).
- **Framework Backbone:** the backbone (Fig. 5 left) has a metaclass parser and C++ interpreter and can dynamically load application libraries with the metaclasses and the C++ compiled classes they extend. The loaded components are shown as icons in a visual browser and are automatically given GUIs reflecting their input and output types. When a component input is changed, the backbone traverses the network passing the data from outputs to inputs and calls the components' C++ update methods, via the C++ interpreter.
  Computer simulation and visualisation is a good example of an application domain where uses can and should easily change roles, as the entire pipeline from CD to and including the EU deals with an experimental area, where neither the algorithms (mapped to components), the application (mapped to the components' assembly), or the application settings (mapped to the component parameter run-time values) are stable or fixed for long periods. The challenge here is, as also in the previous example, to allow for an

easy role transition, and the solution is similar, i.e. user interface tools supporting the run-time control, assembly, and (in the previous example) definition of components.

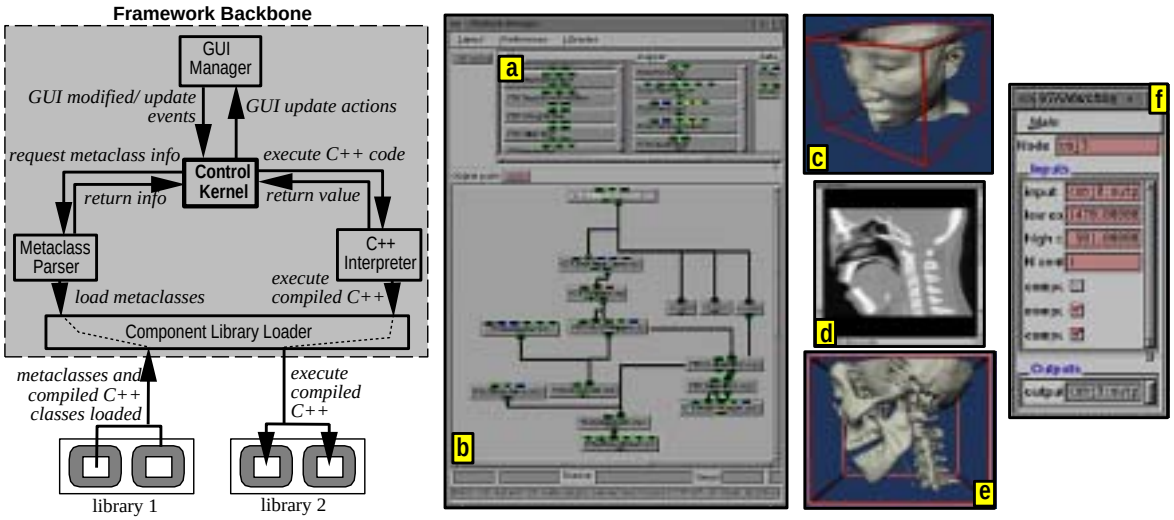


Figure 5. Visualization/simulation framework backbone

This framework falls between single and dual-language systems, as it uses a mix of compiled and interpreted C++ code and metaclass interfaces. Its main novelty is its flexible way to merge compiled/interpreted C++ with dataflow concepts thus combining C++'s developer-level advantages with end-user-level advantages such as interactivity and visual dataflow programming. In this sense, the systems' hot spots are practically the typing systems of the interpreted C++ and the metalanguage.

6.3 A Framework for Industrial Control Systems

Our third example is a framework for industrial control systems development [10] which allows building multitasking applications for control systems (Fig. 3 b). Different customisable control components (robot control, programmable logic circuits (PLC), numerical control) are provided to control the real devices. Other additional control components (e.g. transfer system control) may be added. The framework's elements are:

- **Framework Backbone:** this coordinates the control system tasks (application programs) and provides a virtual communication interface for its components. These communicate via this interface by reading the current data from the controlled devices and calling methods to send commands to the devices. The backbone exchanges the data with the real devices and provides for running application programs in real time.
- **Basic Components:** these are all the components necessary to control a simple production cell with robot arms, digital devices, and numerical systems. Basic components are adapted to the specific characteristics of the controlled devices (e.g. the robot arm dimensions).
- **Additional Components:** these offer new control methods outside the basic set and may use basic components (e.g. a transfer system component using the PLC basic component.

7: Comparison with Earlier Work

Various authors describe framework design as a reverse engineering similar to pattern mining, by factoring out commonalities from a few domain specific solutions [9]. Once it exists, a framework is used to forward engineer further solutions within the same domain. Role analysis usually makes a clear difference between the FD,AD and EU, seen as acting on different achitectural layers. The FDs and ADs are involved with its architectural details whereas EUs are thought to treat the end product as any other software piece. This has two major limitations. First, there is an assumption that a framework is simply an implementation or design tool. However, the framework architecture's inherent flexibility may be *explicitly* exploited also into the final product. Second, white box frameworks limit their extensibility to CDs and ADs as they rely on code source level tools for extension. A better way would be a black box design carried through to the EU role, yielding a high configurability via a common interface at all development stages and an easy role transition. By role analysis, we recognise that various actors have various requirements targeting the basic framework but that these activities can overlap and communicate. Rather than fixing development phases where a CD or AD close an open framework, we advocate frameworks providing extensible, ideally implementation detail free interfaces for all actors, including EUs. Behaviours may be changed by 'horizontal' meta level interfaces [8](e.g. visual tools or scripting languages).

8: Summary and Conclusions

The presented 'framework for framework design' depends on a 'logical pipeline', which starts at the EU's requirements, passes through the AD's and CD's ones, and finally focuses the burden on the FD. The component interface requirements determine the right 'mix' between compile-time and run-time system parts to be implemented by the FD. Several application domains can thus be satisfied by designing several frameworks, all being in fact instances of the same 'meta-framework'. In each case the FD identifies a unique component interface appropriate to the domain. We can be specific by examples only (e.g. our case studies), as the requirements' range is too large to offer general solutions. The union of the users' requirements however induces essential guidelines for a framework design and implementation with optimal cost/benefits. For example, a need to create new instances at run-time asks for a basic interpreter with type instantiation capabilities. Introducing new types at run-time asks for an interpreter capable of dynamic type-loading. If new types introduce new code at run-time, a run-time system support for dynamic code loading is needed. In the extreme case when the border between compiled and interpreted systems vanishes, we may need incremental/on-the-fly compilers, single hierarchy (meta-type based) languages, etc. An important risk in the backbone design is the wrong or under-evaluation of the user requirements and thus providing a simpler, but too weak implementation. Many framework implementations revolve around the compiled and dynamic component-based architectures shown in Section 5, offering more or few 'ad-hoc' run-time component instantiation, typing, or loading features. However, in many cases the user requirements are just above the flexibility limit offered by such framework mechanisms (but usually never above the flexibility of e.g. a programming language's modelling power). FDs should realize in this case that a backbone implementation conforming with the user requirements might better be a single language one. Numerous FDs prefer however not to incur the implementation complexities or speed drawbacks of single language systems and thus choose for a simpler one. This can however have limitations showing up after the framework backbone is completed and frozen, usually causing complex adapter schemes to be coded atop of the existing backbone. The more 'loosely coupled' the component-framework communication/interface is, the easier is everything for all the user categories, as we limit the strategic design decisions to the backbone alone. Simply put, the FD must implement a backbone that satisfies the component interface(s), derived from the union of the user specifications.

References

[1] G. Booch, *Object-Oriented Analysis and Design*, Benjamin/Cummings, Redwood City, CA, second edition, 1994.

[2] W. W. Royce, *Managing the Development of Large Software Systems: Concepts and Techniques*, Proceedings IEEE WESTCON, Los Angeles, 1-9.

[3] S. Demeyer et al., *Design Guidelines for 'Tailorable' Frameworks*, Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 60-65.

[4] R. P. Gabriel, *Patterns of Software - Tales from the Software Community*, Oxford University Press, New York, 1996.

[5] R. Brun, S. Rademakers *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys.Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[6] R. E. Johnson, *Frameworks = (Components + Patterns)*, Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 39-42.

[7] D. L. Parnas, *On Criteria to be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No.12, Dec. 1972, pp. 1053-1058.

[8] T. Mowbray, R. Malveau, *CORBA Design Patterns*, Wiley, 1997.

[9] D. Roberts, R. Johnson, *Patterns for Evolving Frameworks*, in *Pattern Languages of Program Design 3*, R. Martin et al (eds.), Addison-Wesley, 1998.

[10] H. Schmid, *Design Patterns to Construct the Hot Spots of a Manufacturing Framework*, in *The Patterns Handbook: Techniques, Strategies and Applications*, L. Rising (ed.), Cambridge University Press, 1998.

[11] I. Sommerville, P. Sawyer, *Requirements Engineering: a Good Practice Guide*, John Wiley and Sons, 1997.

[12] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992

[13] J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.

[14] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, *The Application Visualization System: A Computational Environment for Scientific Visualization.*, IEEE Computer Graphics and Applications, July 1989, 30–42.

[15] W. Schroeder, K. Martin, B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1995

[16] A.C. Telea, C.W.A.M. van Overveld, *An Object-Oriented Interactive System for Scientific Simulations: Design and Applications*, int *Mathematical Visualization*, H.-C. Hege and K. Polthier (eds.), Springer Verlag 1998

[17] B. Meyer, *Object-oriented software construction*, Prentice Hall, 1997

[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[19] C. Gunn, A. Ortmann, U. Pinkall, K. Polthier, U. Schwarz, *Oorange: A Virtual Laboratory for Experimental Mathematics*, Sonderforschungsbereich 288, Technical University Berlin. URL http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html